A Look at Algebraic Specifications (Mathematical Studies of Information Processing)

AUTHOR(S):

ZILLS, STEPHEN N.

数理解析研究所講究録
454巻 1982年 172-194

172

A Look at Algebraic Specifications

Stephen N. Zilles

Computer Science Department
IBM San Jose Research Laboratory
San Jose, CA 95193/USA

June 11, 1981

Abstract: Being able to specify exactly the properties required of the data used by an abstract algorithm is important in defining highly reusable routines. This paper begins with an introduction to the use of algebras in specifying data types. The notion of what a many sorted algebra is and why they are useful in defining data is reviewed Then the two major approaches to specifying a class of algebras are discussed. Finally, a language for writing well structured specifications is presented with numerous examples. This language is being used in a project that is defining and designing an implementation for highly reusable program units.

WHY AN ALGEBRAIC VIEW

Very loosely, the process of programming can be defined as specifying a *sequence* of *operations* that are applied to relevant *objects* for the purpose of accomplishing some goal. In this definition the important words are *sequence*, *operations* and *objects*. Algebras treat the last two: operations on objects. Sequencing can be treated separately using a technique for describing sequences of actions. For example, one might use flow expressions {Shaw}, path expressions {Haberman and Campbell} or an equivalent technique.

With this view, the specification or description of a problem is divided into two parts: (1) the sequencing of operations and (2) the meaning of the sequences of operations applied to objects. This leads to the factoring of programs, or more accurately the algorithms on which they are based, into two parts. The *schema* which specifies the sequencing of operations and the *interpretation* which specifies the binding of the symbols representing operations and objects to actual programs for operations and representations for objects. With this factorization, the schema is free from being constrained by any particular choice of representation of the objects it acts upon. Thus, it is more truly reusable in the factored form.

With such a factorization, it is important to carefully describe the requirements that are to be placed upon an interpretation of a schema for the interpretation to produce a correct version of the algorithm. From the above discussion, it is clear the interpretation is an algebra and that what is needed is a specification of the class of algebras that correctly interpret the algorithm schema.

WHAT IS AN ALGEBRA

Informally, an algebra has been characterized as a structure in which there is a collection of objects and there are operations upon these objects. Becoming more formal, the collection of objects is separated into a family of disjoint sets, each called a *carrier*. Each carrier is identified by associating with it a *sort* symbol s. The symbol S represents the family of all sorts participating in the algebra. There are, similarly, operations on the carriers which are identified by *operation symbols* σ. Each operation is a mapping from zero or more of the carriers into some one carrier. Thus, each operation symbol is typed with the sort symbols for its operands and its result. The symbol Σ represents the family of all operation symbols and $\Sigma_{w,s}$, where $w=s_1...s_n$, represents all the operation symbols for operations taking sorts $s_1,...,s_n$ into sort s. Σ is the union of all the $\Sigma_{w,s}$.

Algebras can be grouped into classes according to the properties they possess. We will be primarily interested in classes of algebras in which all the algebras have the same set of sort and operation symbols. The set of sort and operation symbols is called the *signature* of the algebra. Through an abuse of notation, we will let Σ stand for the signature and we will let $\underline{Alg}_\Sigma$ denote the class of algebras with the signature Σ. An *algebra* A in $\underline{Alg}_\Sigma$ consists of a family of sets, one for each sort symbol in S, and a collection of operations, one for each operation symbol σ in Σ. The carriers of A will be denoted by s_A for s in S and the operations will be denoted by σ_A for σ in $\Sigma_{w,s}$.

Examples of Algebras

As an example, consider the class of algebras in which the sorts are Booleans, Integers and Sets; and the operations are CREATE a Set, INSERT an Integer into a Set, REMOVE an Integer from a Set and ask if a Set HAS an Integer as a member. In this class, S = {Bool, Int, Set} and $\Sigma_{\lambda,Set}$ = {CREATE}, $\Sigma_{Set\ Int,Set}$ = {INSERT, REMOVE} and $\Sigma_{Set\ Int,Bool}$ = {HAS}. There are many different algebras in this class; the following are three examples:

1) Expected Integer Sets: SET-1

SORTS:

The sets in the carrier are given either by (partial) tabulations of the elements of the set or by reference to some set construction.

$Bool_{SET\text{-}1}$ = {True, False}
$Int_{SET\text{-}1}$ = {0, 1, -1, 2, -2, ...}
$Set_{SET\text{-}1}$ = { {}, {0}, {1,2}, ...}
the normal set-theoretic notion

OPERATIONS:

The operations are given in the form of (very partial) tabulations of the pairs making up the function defining the operation on the carriers.

$HAS_{SET\text{-}1}$ = { < <{},0>, False >,
< <{0},0>, True >,
< <{0},1>, False >,
< <{0,1},1>, True >,
... }

and similarly for the other operations.

2) An array based representation of Sets: SET-2

SORTS:

$Bool_{SET\text{-}2}$ = {1, 0}
$Int_{SET\text{-}2}$ = {0, 1, -1, 2, -2, ..., MaxInt, MinInt}
$Set_{SET\text{-}2}$ = Flexible Arrays of Int
where the elements in the set are the elements in the array
and the flexibility of the array allows it to grow and shrink
with the number of elements currently in the set.

OPERATIONS:

$HAS_{SET\text{-}2}$ = { < <Empty__Array,0>, False >,
< <Extend(Empty__Array,0),0>, True >,
< <Extend(Empty__Array,1),0>, False >,
< <Extend(Extend(Empty__Array,0),1),1>, True >,
< <Extend(Extend(Empty__Array,1),0),1>, True >,
... }

and similarly for the other operations.

This algebra is typical of the algebras used to represent data structures in a computer. The elements of the carriers for each sort are represented by previously defined computer representable structures. In this case, the structures used are bits, fixed

length integers and arrays which can grow and shrink in length using Extend and Contract operations.

3) A very small algebra in Sets: SET-3

SORTS:

$Bool_{SET\text{-}3}$ = {False}
$Int_{SET\text{-}3}$ = {0}
$Set_{SET\text{-}3}$ = { {} }
The carrier for each sort is a single element set.

OPERATIONS:

$HAS_{SET\text{-}3}$ = { < <{},0>, False > }

and similarly for the other operations.

This algebra does not implement our expected notion of sets of integers. It is, however, in the class of algebras with the same collection of sorts and operations as the intended integer set algebras. It is the simplest algebra in that class. The carrier for each sort has only one element, and so, consequently, does the tabulation for each operation.

Algebras with Similar Properties

Do the operands of binary operations commute? When do two expressions created by composing operations compute the same result for all possible operands? These are questions about the structure of an algebra. This structure, to the extent an algebra has any, is embedded in the mappings that define the operations. The behavior of the operations determines the properties possessed by an algebra. These properties can be used to group the algebras into classes of similar algebras. This will be developed in more detail below.

To identify similar algebras we need a method for comparing two algebras. Since we are interested in the similarity of the structure of the algebras, it is natural to consider structure preserving maps as the way to relate two algebras. A *homomorphism* of two algebras, A and B, is a collection of mappings, one for each sort s, such that the behavior of the operations (i.e., the structure of the algebra) is preserved. Notationally, if h:A → B is a homomorphism then for all σ in Σ, for example, σ in $\Sigma_{w,s}$,

$$h(\sigma_A(a_1,...,a_n)) = \sigma_B(h(a_1),...,h(a_n))$$

Using the algebras illustrated above, a homomorphism from SET-1 (ordinary integer sets) to SET-2 (integer sets represented by arrays) would have

h: $Set_{SET\text{-}1}$ → $Set_{SET\text{-}2}$=Array of Integers
h: $Int_{SET\text{-}1}$ →$Int_{SET\text{-}2}$=Fixed Integers
h: $Bool_{SET\text{-}1}$ → $Bool_{SET\text{-}2}$

and

$$h(HAS_{SET\text{-}1}(s,i)) = HAS_{SET\text{-}2}(h(s),h(i))$$

Two algebras are behaviorally identical, that is identical up to their representation, if there is a bijective (one to one and onto) homomorphism between them. (The inverse of the homomorphism is also a homomorphism and their composition is the identity mapping.) Such homomorphisms are called *isomorphisms*.

WHY AN ALGEBRAIC VIEW, PART 2

A second major reason for focusing on an algebraic view of the data being processed by programs is that it naturally frees one of a dependence on the choice of data representation. This *representation independence* comes in three ways. First, a given representation medium, say arrays, can be used in several ways to represent a carrier of a given class of algebras. Secondly, a single representation medium may be used to represent algebras in different classes. Finally, many different representation media can be used to define a class of isomorphic algebras.

Multiple Representations in a Single Medium

With respect to the first point, consider the representation of Integer Sets using Arrays of Integers for the Set carrier. There are several different ways to represent the elements of the Set with an Array.

(SET-2a) $Set_{SET\text{-}2}$ = Array of Int with repeated elements

$$\{1,7,3\} = <1,7,1,3,3,7,1>$$

(SET-2b) $SET_{SET\text{-}2}$ = Array of Int without repetitions

$$\{1,7,3\} = <1,7,3>$$

(SET-2c) $SET_{SET\text{-}2}$ = Ordered Arrays of Int without repetitions

$$\{1,7,3\} = <1,3,7>$$

Although these representations are distinct, by properly defining the mappings for the operations and by defining an congruence relation on the Arrays, the Set algebras they define can be made to be isomorphic. A congruence relation is an equivalence relation that is preserved by the operations. For example, in the unordered representations, Arrays with the same elements but in a different order would be congruent in the algebra. Strictly speaking, when an congruence relation is introduced, the elements of the Set algegra are congruence class, but it suffices to consider the behavior of any element of the congruence class because all elements in the class will behave the same way under the operations

The computational complexity of the operations HAS, INSERT, and REMOVE depends on which of the representations is chosen. Typically, the representation chosen would depend on minimizing the total complexity of the application requiring Sets of Integers. The relative computational difficulty for the above representational approaches is given in the following table.

Operations:	INSERT	HAS	REMOVE
Representation			
SET-2a	Easy	Medium	Hard
SET-2b	Medium	Medium	Medium
SET-2c	Harder	Easier	Harder

TABLE: Computational Difficulty of Set Operations

If INSERTions and REMOVals were relatively infrequent compared to membership tests (HAS) then SET-2c would be used. If, on the otherhand, INSERTions were frequent compared to membership tests, then SET-2a would be used. SET-2b might be used when the frequency of all three operations was comparable.

Several Algebras on a Single Representation Medium

As well as using Arrays to represent Sets, Arrays can also be used as a representation medium for Queues, Sequences and Matrices. Thus choosing a representation does very little to

determine the algebra being represented. It is the mappings for the operations (or the algorithms that implement the mappings) that determine the algebra.

Multiple Representation Media

Since an algebra is fully characterized by the carriers for the sorts and the corresponding mappings for the operations, many different representations for the carriers are possible. Above, we saw several distinct representations for the Set carrier in terms of Arrays. Other possible representations for the Set carrier are Lists (of the elements) and Bit Strings representing the characteristic vector for a set over a domain of finite cardinality.

SPECIFICATION AND CLASSES OF ALGEBRAS

As the above discussion indicates, there is a large number of algebras, even if attention is restricted to the algebras having a particular set of sort and operation symbols. Our interest in algebras is not so much at the level of these individual algebras; it is in classes of similar algebras. For these, it is possible to abstractly represent the properties possessed by all members of the class. Typically this abstraction is expressed in terms of a specification for the algebras that are to belong to the class.

Some of the properties that make a technique good for specifications for algebras used in computer systems are:

1) The specifications are *constructable*; They can be written down in a fairly straight forward way knowing the properties that the class of algebras being specified should have.

2) The technique should have an associated *proof methodology*; this allows one to prove that a given algebra meets the specifications. It also allows properties of the class of algebras to be developed. This includes proving that uses of members of the class correctly implement the algorithm they are used in as well as proving the equivalence or inclusion of specifications.

3) The specifications should be free from *implementation bias*; they should not imply properties that are not intended for the class of algebras being specified. Such unintended properties often occur when extra information that is related to a visualized implementation is put in the specification.

4) The specifications should be *readable* and *testable*; the intended users of the specifications should be able to read them and understand what is being specified. It should also be possible to test if the intended concept has been captured in the specification. Such testing might be be simulation, or exploring the consequences of the specifications.

5) Finally and optionally, the specifications should be *executable*; this allows simple simulation tests of the correctness with respect to the problem requirements of the specification. It also provides a quick and dirty implementation of an algebra in the specified class.

Having given the characteristics that a specification technique should have, there are two main approaches to the writing of specifications:

1) *Model Theoretic* specification in which a generic instance of the intended class is given, and

2) *Axiomatic* specifications in which a set of properties that the members of the intended class must have are given.

MODEL THEORETIC SPECIFICATIONS

In a model theoretic specification the intended class of algebras consists of all algebras that are isomorphic to a generic member of the intended class. This generic member is called the *model.*

An example of a model for the intended class of Set algebras is the following:

Sets REPRESENTED BY Sequences
WHERE
Create() = Empty Sequence
Has(s,e) = IF e=First(s) THEN True ELSE Has(Rest(s),e)
Insert(s,e) = IF Has(s,e) THEN s ELSE Append(s,e)

Here it is assumed that the reader of the specifications is already familiar with the operations First, Rest and Append on Sequences. Therefore, the only problem is to understand the behavior of the operations on Sets that are implemented in terms of the Sequence representation of Sets.

The major problem with model theoretic specifications is that one of two forms of implementation bias can creap into the specifications. First, as above, each abstract element may have more than one representation; for example the set {1,3} is represented by the sequences <1,3> and <3,1>. This problem may be corrected by either being more careful in implementing the operations or by defining an explicit equality relation on the representation.

The second form of implementation bias is more subtle. This occurs when details of the implementation need not be preserved to have correctly implemented the intended abstraction; for example, one needn't be able to inquire about the order of elements in a sequence to have correctly implemented the Set abstraction. This form of bias is more difficult to detect.

AXIOMATIC SPECIFICATIONS

In contrast to the model theoretic specifications which are very concrete, the axiomatic specifications characterize the properties that the abstract class is intended to have. In the common approach {ZILS74, THAJ78}, the axiom are expressed in equational form. These equations are something like simplification rules that specify when two sequences of operations yield the same result. A sample specification for the intended class of Set algebras is:

Set **IS**

SORTS: S, E, B OPERATIONS: Create: → S
Insert: SxE → S
Remove: SxE → S
Has: SxE → B

AXIOMS: Has(Insert(s,i),j) = IF i=j THEN True ELSE Has(s,j)
Has(Create,j) = False
Remove(Insert(s,i),j) = IF i=j THEN Remove(s,j) ELSE Insert(R
Remove(Create,j) = Create
Insert(Insert(s,i),i) = Insert(s,i)
Insert(Insert(s,i),j) = Insert(Insert(s,j),i)

These axioms may be satisfied in many algebras so something more must be said about what class of algebras are specified by an axiomatic specification. Here we consider only two possible interpretations of the class specified. One interpretation is the the class specified consists of all the algebras in which the axioms hold. This approach has the disadvantage that the class has non-isomorphic models within it. That is two algebras can satisfy the specifica-

tion but have very different behavior where the axioms dont require a particular behavior. This is all right if only the properties directly stated in the axioms are needed to establish the correctness of the use of the algebra in some context. This is frequently the case when considering the parameters of a parameterized specification.

The other interpretation of the class defined by the specification is that it is the subclass of initial algebras within the larger class. An algebra is *initial* if there is a unique homomorphism from that algebra to every other algebra in the larger class. It is not difficult to prove that all initial algebras are isomorphic. With this fact, the initial algebra interpretation defines an algebra that is unique up to isomorphism. It also has the property the two terms in the operations of the algebra are equal only if forced to be by the equational axioms. Or, in model theoretic term, only if the terms are equal in all models. In the sequel we will use both interpretations of the axioms.

MODEL THEORETIC VS AXIOMATIC SPECIFICATIONS

The strengths of the model theoretic approach are that you know the specification is consistent if it can be constructed at all, the model can give useful hints to the implementer, and the resulting algebra is more likely to computable than an axiomatic specification. The strengths of the axiomatic approach is that there is no implementation bias and it is easier to establish properties of the algebra, such as the correctness of a use of the algebra. This suggests that the appropriate methodology is develop both a model theoretic and an axiomatic definition and prove their equivalence.

STRUCTURED SPECIFICATIONS

Specifications, whether in axiomatic or model theoretic form, are normally developed in the context of previous relate work. It is fairly rare that specifications are developed completely from scratch. For example, the above specification for Sets assumed the existence of specifications for Booleans and the Set elements. To make the process of writing specifications manageable, it is necessary to build on previous work rather than to redo it. This improves productivity; it also simplifies the process of understanding a specification because a major portion of the specification is in terms of already understood concepts.

This section introduces a language for axiomatic specifications that allows a complex specification to be built up from a number of simple specification steps. This language is similar to and is, in part, derived from the specification languages developed by Burstall and Goguen

{BURS77, BURS80} and by Hupbach, Kaphengst and Reichel {HUPB80}. The relationship to these languages will be discussed later in this section. Our language also has roots in a structured specification language proposed by Ehrich and Thatcher {EHRD81}; this in turn was motivated by the Language for Computer Algebra developed by Jenks, Davenport and Trager {JENR81}. The algebra of parameterization was first discussedd by Thatcher, Wagner and Wright in {THAJ78}.

The basic building block of this language is the *specification expression*. Specification expressions can be combined to denote a *specification*. A specification is a statement which identifies:

1) the sorts the participate in the class of algebras being specified;

2) the operations applicable to the elements of the carriers corresponding to the sorts;

3) axioms which constrain the behavior of the operations; and

4) constraints on class of algebras which are to satisfy the specifications.

The first three aspects of the specification have already been introduced. The sorts and operations define the syntax of the class of algebras and the axioms and constraints define the semantics of the class. The basic purpose of the constraint is to specify when the class of specified algebras is to be limited by allowing only those algebras in which some subalgebras are initial (or freely generated). More on this below.

Basic Specification

The simplest form of specification is one with no constraints. There is a corresponding specification expression in which the sorts, operations and axioms for the specification are introduced explicitly. This has the form:

<spec-expr> ::= SORTS: <sort-list>
OPERATIONS: <operation-list>
AXIOMS: <axiom-list>

All three clauses are optional. Since this is an informal presentation of the language, the forms of the various lists will not be given in detail but should be obvious from the examples given below. The specification associated with this expression has the sorts and operations

listed and is constrained by the axioms. The class of algebras specified are all algebras having the listed signature and for which the axioms hold. This includes but is not limited to the algebras which are initial in this class.

The following is a very simple example of a basic specification:

Triv **IS**

SORTS: S

The specification simply defines an algebra with one sort and (possibly) no operations. Any algebra having a single sort S satisfies this specification.

This example also illustrates another aspect of the specification language. Specifications can be given names for use in specification expressions. Here the name *Triv* is given to the specification that results from the specification expression following **IS** The general form of the naming clause is:

<defn> ::= <name> **IS** <spec-expr>

This <name> can be used in place of the specification it names:

<spec-expr> ::= <name>

The names of specifications will be printed in italics as was done for *Triv* above.

Simple Constructions

If the intent of the specification is to restrict the class of algebras satisfying the specification to the set of initial algebras, then a constraint must be added to the specification. This is specified in the language using a simple construction expression:

<spec-expr> ::= **CONSTRUCTING** <spec-expr>

The first three parts of the specification given by this clause are the same as determined by the righthand <spec-expr>. To this is added, however, a constraint that only the initial algebras in the class satisfying the righthand specification are in the class satisfying the lefthand specification.

As examples of simple constructions, consider the following:

Bool1 **IS CONSTRUCTING**

SORTS: B

OPERATIONS: True: -> B

False: -> B

Nat1 **IS CONSTRUCTING**

SORTS: N

OPERATIONS: Zero: -> N

Succ: N -> N)

In these two specifications, there are no axioms to satisfy so the initial algebras are simple the term algebras in the given operations. For *Bool1* there are only two terms corresponding to the two constant operations True and False. For *Nat1* there is an infinite set of terms beginning with Zero and successively prefixing the previous term with a new occurrence of Succ {Zero, Succ(Zero), Succ(Succ(Zero)), ...}. The constraint eliminates finite and nonstandard models from the class being specified.

Extensions

The purpose of extensions to specifications is to require additional properties of the algebras that are to satisfy the specification. The extension specification can be used

1) to define additional operations that must exist within the class of algebras being specified, and

2) to subset the class to those algebras having certain properties expressed as axioms.

The form of a specification extension is:

<spec-expr> ::= <spec-expr> **WITH** <spec-expr>

The specification defined by this expression is the union of the sorts, operations, axioms and constraints of each specification respectively. The algebras satisfying the resulting specification are those with the union signature and satisfy the union of the axioms and the constraints.

One use for the extension clause is the require the existence of new operations which are given an axiomatic definition. This approach is used to extend *Bool1* and *Nat1* to have a more complete set of operations:

Bool **IS** *Bool1* **WITH**

OPERATIONS:	and:	B x B	-> B
	or:	B x B	-> B
	not:	B	-> B

AXIOMS:

and(b,True) = b
and(b,False) = False
or(b,True) = True
or(b,False) = b
not(True) = False
not(False) = True

Nat **IS** *Bool* **WITH** *Nat1* **WITH**

OPERATIONS:	plus:	N x N	-> N
	mult:	N x N	-> N
	lteq:	N x N	-> B

AXIOMS:

plus(n,zero) = n
plus(zero,n) = n
plus(succ(n1),n2)) = succ(plus(n1,n2))
mult(n,zero) = zero
mult(zero,n) = zero
mult(succ(n1),n2) = plus(n1,mult(n1,n2))
lteq(zero,n) = True
lteq(succ(n1),succ(n2)) = lteq(n1,n2)
lteq(succ(n),zero) = False

The equational axioms act like recursive equation definitions for the additional operations that are to be possessed by the algebras satisfying the specifications. Both of these specifications are based on a simple construction so the class of algebras is limited to the algebras which are initial with respect to the signatures of *Bool1* and *Nat1*. This class is then further refined to require that the algebras in the specified class also have the operations defined immediately above.

Note that *Nat* is based on both *Nat1* and *Bool*. Hence, the specification for *Nat* is the union of the specifications for *Bool*, *Nat1* and the explicit basic specification. There are two constraints in the specification for *Nat*: one for *Bool1* and one for *Nat1*. Adding to the specifications does not change the portion of the specification covered by a constraint. Therefore, the sorts B and N must still be initial with respect to the simple signatures of *Bool1* and *Nat1*.

To see the second function of the extension mechanism we develop a traditional algebraic example in the language. First, we define a relatively trivial class of algebras that represent sets of elements that are to be structured:

Elem **IS** *Bool* **WITH**

SORTS:	E
OPERATIONS:	Eq: E x E → B
AXIOMS:	Eq(e,e) = True
	Eq(e1,e2) = Eq(e2,e1)
	Eq(e1,e2)=True & Eq(e2,e3)=True => Eq(e1,e3)=True

This specification has one sort and there is an equality operation on the elements of the corresponding carrier.

Given this set of elements, we develop specifications for the classes of semigroups (sets with one associative operation), monoids (semigroups that have a unit or identity element), and semigroups and monoids that are free with respect to a set of generating elements.

Semiaxs **IS**

OPERATIONS:	*: S x S → S
AXIOMS:	(s1*(s2*s3)) = ((s1*s2)*s3)

Semigrp **IS**

Triv **WITH** *Semiaxs*

This extension restricts the class of algebras with a sort S to those which also have an associative binary operation '*'.

Monoaxs **IS**

OPERATIONS: 1 → S

AXIOMS: 1*s = s*1 = s

Monoid **IS**

Semigrp **WITH** *Monoaxs*

By adding the *Monoaxs* to the specification for *Semigrp*, the class of algebras satisfying the union of the two specifications is further restricted to algebras with a sort S having an associative operation '*' and a unit '1'. Thus, each extension potentially reduces the size of the class of algebras satisfying the specification.

More Complex Constructions

The construction of the free semigroup over a set is typical of constructions that use some (sub-)algebra as the basis for constructing a new algebra. Again, it is desirable that the algebras being constructed are the initial ones so that the construction is unique up to isomorphism. In this case, the algebras are to be initial in the class of algebras having an image of the (sub-) algebra that is the basis for the construction.

More formally, let there be two specifications *A* and *B* such that *B* is contained in *A*. An algebra A satisfying *A* is a *free extension* of an algebra B satisfying *B* iff every *B* homomorphism h_B:B→C where C is an arbitrary *A* algebra extends uniquely to a *A* homomorphism h_A:A→C. Thus, in this sense, the algebras that are free extensions of the given *B* algebra B are initial in the class of all *A* algebras with images of B. Hence, these algebras are all isomorphic and are unique up to isomorphism. Typically, although not necessarily, we will require that the image of B in the construction be isomorphic to the basis algebra B. Such constructions are called *persistent*.

This more complex construction takes two specification expressions as operands. Linguistically, this takes the form:

<spec-expr> ::= <spec-expr> **CONSTRUCTING** <spec-expr>

where the middle <spec-expr> specifies the class of possible basis algebras for the construction and the rightmost <spec-expr> specifies what is to be constructed. The resulting specification is the union of the two specification plus a new constraint between the specifica-

tion to the left of **CONSTRUCTING** and the specification to the right of **CONSTRUCTING.** The simple form of the construction simply has an empty basis algebra. An algebra satisfies the combined specification if it satisfies the first three parts of the specification and the set of constraints including the free extension constraint for this construction is satisfied.

As examples of more complex construction, we look at the construction of the free semigroup and free monoid on a set of generators:

Freesemigrp **IS**
Elem **CONSTRUCTING** (*Semigrp* **WITH**
OPERATIONS: inj: E → S)

The sort E of *Elem* designates the carrier for the elements that can be multiplied together in the free semigroup. Using concatenation to represent multiplication, the elements of the semigroup S can be viewed as being all possible concatenations of the elements of E; for example, {e1, e2, e1e2, e2,e3, ..., ei1ei2...ein, ...}. The empty concatenation is not in this algebra.

If the operation 'inj' which injects the basis set into the carrier for the semigroup did not exist, the semigroup would be empty since there would be no operation to force any values into the sort S of the semigroup. The above specification for *semigrp* does not involve a constraint so that specification encompasses all semigroups, whether free with respect to a given basis set of elements or not. It is the constraint that limits the satisfying algebras to empty semigroup when the *Elem* set is empty.

A similar construction specifies the free monoid on a set of generators in *Elem*:

Freemonoid **IS**
Elem **CONSTRUCTING** (*Monoid* **WITH**
OPERATIONS: inj: E →S)

In this case, the empty concatenation is now in the constructed set S because it is the unit required by the axioms for a monoid.

These examples clearly show that the specification expressions for construction and extension have very different results. A slight variation on these examples shows that these two operations are not mutually associative. The specification for *Freemonoid* can be rewritten as:

Freemonoid **IS**

Elem **CONSTRUCTING** (*Semigrp* **WITH** *Monoaxs* **WITH**
OPERATIONS: inj: E →S)

where the definition of *Monoid* has been expanded into its constituent parts. Now consider what happens when the parentheses are slightly rearranged.

Makemonoid **IS**

(*Elem* **CONSTRUCTING** *Semigrp* **WITH**
OPERATIONS: inj: E →S) **WITH** *Monoaxs*

The regrouping of the parentheses eliminates the monoid axioms from the construction. Hence, the construction is the same as that for *Freesemigrp*. Then, the additional extension to include the *Monoaxs* subsets the class of free semigroups to those that have units. But, as the above analysis to the *Freesemigrp* construction shows, the empty concatenation is not in the free semigroup and there is no unit. Therefore, the class specified is the empty class which is certainly different from the class of free monoids.

Adaptation

The above examples show how a specification can be built up from small pieces. There are cases, however, when the specification that is desired is only slightly different from an existing specification. A small number of "changes" to the existing specification are all that is needed to create the desired specification. The specification language provides a way to describe how these changes are to be made. This is called *adapting* a specification:

<spec-expr> ::= <spec-expr>[<repl-list>]

<repl> ::= <spec-expr> **FOR** <spec-expr>
<sort> **FOR** <sort)
<opr> **FOR** <opr>

An adaptation causes the <spec-expr> (or portion thereof) that is on the left of the **FOR** to replace righthand side <spec-expr> (or portion thereof) within the original specification. A replacement is legal iff there is a mapping from the sorts, operations, axioms and constraints of the replaced subspecification into the sorts, operations, axioms and constraints of the replacement. The modified specification is the result of this clause.

A simple example of adaptation obtains the set of *Sequences* from the specification for *Freemonoid* by renaming operations:

Sequence **IS**

Freemonoid[Concat **FOR** *, Empty for 1]

Similarly, the algebra for *Elem* could have been obtained from the algebra for *Triv* by renaming the sort symbol and adding the appropriate axioms.

Elem **IS** *Triv*[E **FOR** S] **WITH** *Bool* **WITH**

OPERATIONS:	Eq:	E x E → B
AXIOMS:	Eq(e,e) = True	
	Eq(e1,e2) = Eq(e2,e1)	
	Eq(e1,e2)=True & Eq(e2,e3)=True => Eq(e1,e3)=True	

Using the capability to upgrade entire subspecifications, extensive changes to a specification can be achieved. For example, it is possible to adapt the notion of a sequence to sequences of elements with a binary operation on the element carrier. This specification can then be used to define a Reduce operation that combines all the elements in a sequence into a single value.

BinElem **IS** *Elem* **WITH**

OPERATIONS:	Binop:	E x E → E
	Id:	→ E
AXIOMS:	Binop(e,Id) = Binop(Id,e) = e	

SequenceD **IS**

Sequence[*BinElem* **FOR** *Elem*]

Here the extended *BinElem* replaces the original subspecification for *Elem* in the *Sequence* specification.

SequenceR **IS** *SequenceD* **WITH**

OPERATIONS:	Reduce: S	→ E
AXIOMS:	Reduce(Empty) =Id	
	Reduce(s1 Concat s2) = Binop(Reduce(s1),Reduce(s2))	

The adaptation clause can then be applied to define sequences of *Nat*ural numbers in which the binary operator for combinations is addition ('+'). This involves a double adaptation. The first step is to change the names of the operations used in the *SequenceR* specification to correspond to the desired operations on natural numbers and then the specification *Nat* is substituted for the generic *BinElem* specification.

NatSequence **IS**

SequenceR[+ **FOR** Binop, 0 **FOR** Id][*Nat* **FOR** *BinElem*]

Using adaptation to substitute one (sub-) specification for another is only possible when the replacement specification is a restriction on (implies) the original (sub-) specification. For example, suppose we define

ComMonoid **IS** *Monoid* **WITH**

AXIOMS: s1*s2 = s2*s1

as the specification for commutative monoids. Then it is not legal to substitute

Nat **FOR** *ComMonoid*[plus **FOR** *, zero **FOR** 1]

since the commutivity axiom was not explicitly defined in the specification for *Nat*. But, it is clear that that axiom is consistent with the specification for *Nat* so the following substitution would be legal.

Nat **WITH** AXIOMS: plus(n1,n2) = plus(n2,n1)
FOR *ComMonoid*[plus for *, zero for 1]

Thus, specifications which are not directly mappable one to another but are logically equivalent can be brought into harmony by appropriate explicit definitions for the missing operations and axioms.

Comparison with Other Work

The language proposed here is most closely related to the specification language of Hupbach, Kaphengst and Reichel {HUPB80}. What I call a specification, they call a *canon*. Their approach is developed for partial algebras, called *equoids*, and this approach assumes that the algebras are total. They use the term *initial restriction* for what I have called a constraint. These are not important distinctions, however. The main distinctions are, first, the elimination

of a separate construct for joining specification together. In the approach used here, the extension clause handles both combinations of specifications and extensions of specifications. The second distinction is the use of an adaptation clause instead of canon macros. The canon macros are paramaterized specifications in which the parameter is a subspecification that can be replaced to instantiate a particular specification. This mechanism is not as flexible as the adaptation mechanism because it forces one to choose ahead of time which portions of the specification can be replaced. This is not necessary with the adaptation mechanism.

The above comments apply to the work of Burstall and Goguen as well. Burstall and Goguen define *theories* rather than specifications. The theory is the specification in which all the implications of the axioms have been explicitly added. This makes it simpler to replace one theory with another, but, as the above discussion at the end of the section on adaptations shows, it is not difficult to extend specifications to match identically when they would have the same theories. Burstall and Goguen use the term *data constraint* for what I call a constraint. They have also not considered exactly what class of algebras is defined by one of their theory definitions.

ACKNOWLEDGEMENTS

The work reported in this paper was done in collaboration with Peter Lucas and Jim Thatcher. Jim Thatcher read portions of the paper and provided many useful comments. The author is most grateful for the stimulating interchanges with these people. They should not in any way be held responsible for any errors in the paper; for those I take total responsibility.

REFERENCES

<BURR77> Burstall, R. and Goguen, J., "Putting Theories Together to Make Specifications," *Proc. Fifth International Joint Conference on Artificial Intelligence,* (August 1977), 1045-1058.

<BURR80> Burstall, R. and Goguen, J., "An Informal Introduction to Specifications Using CLEAR," SRI Report, Menlo Park, Ca, December 1980.

<CAMR74> Campbell, R. H. and Habermann, A. N., "The Specification of Process Synchronization by Path Expressions," in *Lecture Notes in Computer Science,* Vol 16: *Operating Systems,* Springer-Verlag, 1974, pp 89-102.

<EHRH81> Ehrich, H.-D. and J.W. Thatcher, "A Structured Specification Language to Reflect the Organization of the NEWSPAD Language," IBM Research Report, in preparation, 1981.

<GOGJ77> Goguen, J.A., Thatcher, J.W., Wagner E.G., and Wright, J.B., "Initial Algebra Semantics and Continuous Algebras", *Journal ACM 24* pp. 68-95 (January 1977).

<HUPB80> Hupback, U. L., Kaphengst, H. and Reichel, H., Initial Algebraic Specifications of Datatypes, Parameterized Datatypes and Algorithms, VEB Robotron ZFT, Tech. Report, Dresden, 1980.

<JENR81> Jenks, Richard D. and Barry M. Trager, "A Language for Computer Algebra," Proceedings of the 1981 Symposium of Symbolic and Algebraic Computation, August, 1981.

<SHAW79> Shaw, A. C., "Software Specification Languages Based on Regular Expressions," in *Software Development Tools Workshop Conference,* NBS/NASA, Pingree Park, Colorado, May 1974.

<THAJ78> Thatcher, J.W., Wagner, E.G. and Wright, J.B., "Data Type Specification: Parameterization and the Power of Specification Techniques," in Proc. Tenth Annual ACM Symposium on Theory of Computing, ACM, New York (May 1978), 119-132.

<ZILS74> Zilles,S.N., "Algebraic Specifications of Data Types," *Project MAC Progress Report 11*, Massachusetts Institute of Technology, Cambridge, MA, 1974, 52-58.